

# Towards Logical Architecture and Formal Analysis of Dependencies Between Services

Maria Spichkova and Heinrich Schmidt

Computer Science and IT, RMIT University, Melbourne, AUSTRALIA

Email: {Maria.Spichkova,Heinz.Schmidt}@rmit.edu.au

**Abstract**—This paper presents a formal approach to modelling and analysis of data and control flow dependencies between services within remotely deployed distributed systems of services. Our work aims at elaborating for a concrete system, which parts of the system (or system model) are necessary to check a given property. The approach allows services decomposition oriented towards efficient checking of system properties as well as analysis of dependencies within a system.

**Keywords**—formal methods; static analysis; dependencies between services; decomposition; verification

## I. INTRODUCTION

This approach originated from the analysis of two case studies from automotive area, which were developed together with industrial partners within DenTUM and Verisoft-XT<sup>1</sup> projects. The first case study [1], developing an Adaptive Cruise Control system with Pre-Crash Safety functionality, was motivated and supported by DENSO Corporation, the second case study [2], developing a Cruise Control System with focus on system architecture and verification, was supported by Robert Bosch GmbH. One of the essential questions we have investigated during this analysis was which part of the system functionality do we need to analyse to check a certain property in sense of monitoring, testing or formal verification. In most cases, we don't need complete information about the system as a whole to analyse certain aspects or to check certain properties. The problem is how to find which local information (at the level of one or more components or subsystems) is sufficient for this check, and how to solve this problem on a formal level to allow the automatisisation of the analysis.

The reason for this question has a very practical ground: any additional information about the system can make the whole process slower, more expensive or even infeasible, especially if we are speaking about verification. In the case of model checking this can lead to the state explosion problem. In the case of theorem proving this can lead to stack overflow or excessive or prohibitive amounts of time needed by verification engineers to complete the task.

On the logical level, this problem can be reformulated as follows: What is the minimal part of model needed to check a specific property? We suggest an approach focusing on data and control flow dependencies between services. Dependencies' analysis results in a decomposition that gives rise to a logical system architecture, which is the most appropriate for the case of remote monitoring, testing and/or verification.

In the case where the check of properties depends on information in one location, obviously we can check the property without remote connection and without sending system-wide data for inclusion in the monitoring, testing, or verification process. In the case of remote connection, an additional problem arises through the need of transferring large amounts of data, for example from sensor instruments or automated test generators and model checkers. Thus, it is crucial to analyse regarding to efficiency, which properties (or their parts) should be checked locally, and which should be sent to the cloud. Therefore, an appropriate model covering all these aspects of remotely deployed distributed systems is needed to fulfil the corresponding constraints and to make decisions on (logical) service-oriented system architecture at least semi-automatically.

**Contributions:** In this paper, we present a formal approach that allows modelling of data and control flow dependencies between services. The aim of the approach is to allow decomposition oriented towards efficient checking of system properties, also taking into account such characteristics as performance, worst-case execution time, reliability, etc. Another contribution of this approach is a semi-automatic support of these ideas on verification level. Applying ideas introduced in this paper within specification and proof methodology [3], and taking into account human factor analysis [4], [5], we obtain concise and at the same time readable specifications. To support this approach on verification level, we have built the corresponding set of theories in Higher-Order Logic. It allows to check dependency relations using an interactive theorem prover Isabelle/HOL [6]. To discharge proof goals automatically, we apply Isabelle's component Sledgehammer [7] that employs resolution based first-order automatic theorem provers (ATPs) and satisfiability modulo theories (SMT) solvers.

## II. INTER-/INTRASERVICE DEPENDENCIES

In this paper, we follow the definition of services introduced in [8], where a service  $S$  is defined as a partial function from input streams  $\mathbb{I}(S)$  to output streams  $\mathbb{O}(S)$ . Thus, the behaviour of a service can be defined partially (only for a subset of its possible inputs), in contrast to the totality of a component interface behaviour. In our approach, we need only an abstract definition of a service: we focus on the analysis of dependencies within and between services, where the behaviour of a (sub)services can be specified using different languages.

While modelling communication between services on a certain abstraction level  $L$  (i.e. level of refinement/decomposition), we specify the dependencies by the function

<sup>1</sup> Verisoft-XT project – <http://www.verisoftxt.de/>

$$Sources^L : CSet^L \rightarrow (CSet^L \text{ set})$$

$CSet^L$  denotes here the set of services at the level  $L$ .  $Sources^L$  returns for any service  $A$  the corresponding (possibly empty) set of services that are the sources for the input streams of  $A$ . More precisely, all the dependencies can be divided into the direct and indirect ones. As a *stream* we understand here both data and control flows. On the level of logical architecture, we can see a stream as an abstract *channel*.

Similar to the function  $Sources^L$ , we define direct dependencies by the function

$$DSources^L : CSet^L \rightarrow (CSet^L \text{ set})$$

For example,  $C_1 \in DSources^L(C_2)$  simply means that at least one of the output channels of  $C_1$  is directly connected to some of input channels of  $C_2$ . In more complicated situations, we need to reason about the (transient) dependencies within many or all of the services of the system.

The direct sources for  $C$  can be defined as follows:

$$DSources^L(C) = \{S \mid \exists x \in \mathbb{I}(C) \wedge x \in \mathbb{O}(S)\}$$

If we do not have to take into account the indirect dependencies, the function  $Sources^L$  is defined to be equal to the function  $DSources^L$ , otherwise we define this function recursively over the set of services using the following fixed point construction:

$$Sources^L(C) = DSources^L(C) \cup \bigcup_{S \in DSources^L(C)} \{S_1 \mid S_1 \in Sources^L(S)\}$$

The functions  $DSources^L(C)$  and  $Sources^L(C)$  have a number of crucial properties. For example:

(1) If a service  $X$  does not belong to the  $DSources^L$  set of any service  $C$  of the system, it also does not belong to the  $Sources^L$  set of any service  $C$ :

$$\begin{aligned} \forall C \in CSet^L. X \notin DSources^L(C) &\Rightarrow \\ \forall C \in CSet^L. X \notin Sources^L(C) \end{aligned}$$

Thus, if  $C$  belongs to the  $Sources^L$  set of a service  $S$ , then exist some service  $Z$  s.t.  $C$  belongs to its  $DSources^L$  set (as a special case,  $Z$  could also be equal to  $S$ ):

$$C \in Sources^L(S) \Rightarrow \exists Z. C \in DSources^L(Z)$$

This also imply that if  $Sources^L$  set of a service  $C$  is empty, its  $DSources^L$  set is empty too, and vice versa:

$$Sources^L(C) = \emptyset \Leftrightarrow DSources^L(C) = \emptyset$$

(2) Transitivity: If a service  $C$  belongs to the  $Sources^L$  set of a service  $S$ , which itself belongs to the  $Sources^L$  set of another service  $Z$ , then the service  $C$  also belongs to the  $Sources^L$  set of a service  $Z$ .

$$C \in Sources^L(S) \wedge S \in Sources^L(Z) \Rightarrow C \in Sources^L(Z)$$

(3) If we have mutual dependencies between services, then their  $Sources^L$  sets are equal and contain these services themselves ( $XS$  and  $ZS$  denote here some sets over  $CSet^L$ ):

$$\begin{aligned} Sources^L(C) &= (XS \cup Sources^L(S)) \wedge \\ Sources^L(S) &= (ZS \cup Sources^L(C)) \\ \Rightarrow Sources^L(C) &= Sources^L(S) = XS \cup ZS \cup \{C, S\} \end{aligned}$$

In general, values of an output channel  $y \in \mathbb{O}(C)$  of a service  $C$  do not necessarily depend on the values of all its input streams. This means that an optimisation of system/services' architecture may be needed in order to localise these dependencies. To express any restrictions we use the following notation:  $\mathbb{I}^D(C, y)$  denotes the subset of  $\mathbb{I}(C)$  that  $y$  depends upon. There are three possible cases to consider:

- $y$  depends on all input streams of  $C$ :  $\mathbb{I}^D(C, y) = \mathbb{I}(C)$ ;
- $y$  depends on some input streams of  $C$ :  $\mathbb{I}^D(C, y) \subset \mathbb{I}(C)$ ;
- $y$  is independent of any input stream of  $C$ , i.e.,  $\mathbb{I}^D(C, y) = \emptyset \neq \mathbb{I}(C)$ .

While determining these sets, we should take into account not only the direct dependence between input/output values, but also the dependence via local variables of the service. For example, let  $C$  be a service with a local variable  $st$ , representing its current state, and an output channel  $y$ , which depends only on the value of  $st$ . If  $st$  is updated depending to the input messages the service receives via the input channel  $x$ , then  $\mathbb{I}^D(C, y) = \{x\}$ . To be more precise, we mark this special case by an upper index  $\mathbb{I}^D(C, y) = \{x^{(st)}\}$  indicating that  $x$  influences  $st$ , and  $y$  via  $st$ .

#### A. Elementary Services

Based on the definition above, we can decompose services to have for each output channel the minimal sub service computing the corresponding results (we call them *elementary services*). An elementary service either

- should have a single output channel (in this case this service can have no local variables), or
- all its output channels are correlated, i.e. mutually depend on the same local variable(s).

Let  $C$  be a service with  $m$  input channels  $x_1, \dots, x_m$  and  $n$  output channels  $y_1, \dots, y_n$  as well as  $k$  local variables  $l_1, \dots, l_k$ . For each output channel  $y_i$ ,  $1 \leq i \leq n$  we check the corresponding set  $\mathbb{I}^D(C, y_i)$ :

(1) If  $\mathbb{I}^D(C, y_i)$  contains only direct dependencies from some input channels, i.e.  $\mathbb{I}^D(C, y_i) \subseteq \mathbb{I}(C)$ , we remove the corresponding computations from  $C$  to a single subservice  $C_i$ , which has a single output channel  $y_i$ .

(2) If  $\mathbb{I}^D(C, y_i)$  contains also dependencies via some local variables, then we should check which other output channels depend from these variables, because the output channels depending from the same local variables will belong to the same subservice of  $C$ , otherwise we will need duplicated computation for these variables. Thus, we proceed as follows:

- If  $y_i$  is the only output channel depending from these variables, we remove the corresponding computations from  $C$  to a single subservice  $C_i$ , which has a single output channel  $y_i$ .
- If there are other  $jm$  output channels  $y_{j1}, \dots, y_{jm}$  depending from these variables, we remove the corresponding computations from  $C$  to a single subservice  $C_i$ , which has  $jm + 1$  output channels  $y_i, y_{j1}, \dots, y_{jm}$ .

For all three cases,  $\mathbb{I}^D(C, y_i)$  becomes a set of input channels of the new subservice  $C_i$ . If after this decomposition a single service is too complex, we can apply the decomposition strategy presented in [9].

### B. Running Example

Let us illustrate the presented ideas by an example: we show how each service can be decomposed to optimise the dependencies within each single service, and after that we optimise the architecture of the whole system. Given a system  $S$  (cf. also Fig. 1) consisting of five services, where the set  $CSet$  on the level  $L_0$  is defined by  $\{A_1, \dots, A_5\}$ . The sets  $\mathbb{I}^D$  of data and control flow dependencies between the services are shown in Table I. We represent the dependencies graphically using dashed lines over the service box.

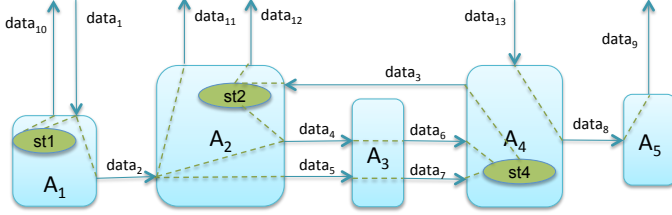


Fig. 1. System  $S$ : Dependencies and  $\mathbb{I}^D$  sets

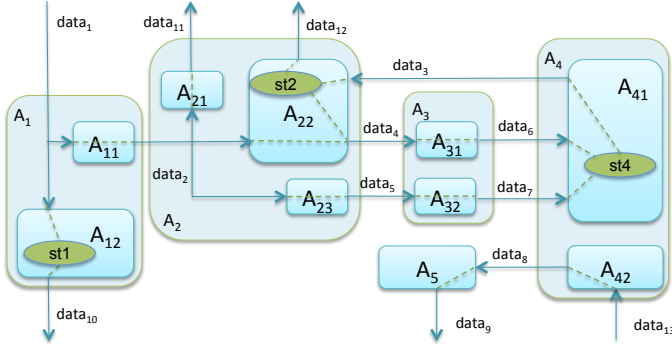


Fig. 2. Services' decomposition (level  $L_1$ )

Now we can decompose the system's services according to the given  $\mathbb{I}^D$  specification. This results into the next abstraction level  $L_1$  of logical architecture (cf. Fig. 2), on which all services are elementary:

- $A_1$  has two output channels,  $data_2$  depending on the input channel  $data_1$  directly and  $data_{10}$  depending on the same input channel but via a local variable  $st1$ . Therefore, it should be decomposed in two subservices:  $A_{11}$  with input  $data_1$  and output  $data_2$ , and  $A_{12}$  with input  $data_1$  and output  $data_{10}$ .
- $A_2$  has four output channels:  $data_5$  and  $data_{11}$  depend on the input channel  $data_2$  directly, the corresponding computations should be removed to two subservice,  $A_{23}$  and  $A_{21}$ , respectively.  $data_{12}$  depends on the input channel  $data_3$  via local variable  $st2$ , but there is another output channel  $data_4$  depending on this local variable, therefore, computations for these two outputs should belong to the same subservice  $A_{22}$ . Thus,  $A_2$  should be decomposed in three subservices.
- $A_3$  has two output channels,  $data_6$  depending on the input channel  $data_4$  directly and  $data_7$  depending on the input channel  $data_5$  directly. Therefore, it should be decomposed in two subservices.

$A_4$  has two output channels,  $data_8$  and  $data_{13}$ . It should be decomposed in two subservices, because  $data_8$  depends on the input channel  $data_{13}$  directly and  $data_{13}$  depends on the input channels  $data_6$  and  $data_7$  via local variable  $st4$ .

$A_5$  has only one output channel, which means that no decomposition on this level is needed.

TABLE I. DEPENDENCIES WITHIN THE SYSTEM  $S$

	$DSources^{L_0}$	$Sources^{L_0}$	$\mathbb{I}^D$
$A_1$	$\emptyset$	$\emptyset$	$data_2 : \{data_1\}$ $data_{10} : \{data_1^{st1}\}$
$A_2$	$\{A_1, A_4\}$	$\{A_1, A_2, A_3, A_4\}$	$data_4 : \{data_2, data_3^{st2}\}$ $data_5 : \{data_2\}$ $data_{11} : \{data_2\}$ $data_{12} : \{data_3^{st2}\}$
$A_3$	$\{A_2\}$	$\{A_1, A_2, A_3, A_4\}$	$data_6 : \{data_4\}$ $data_7 : \{data_5\}$
$A_4$	$\{A_3\}$	$\{A_1, A_2, A_3, A_4\}$	$data_3 : \{data_6^{st4}, data_7^{st4}\}$ $data_8 : \{data_{13}\}$
$A_5$	$\{A_4\}$	$\{A_1, A_2, A_3, A_4\}$	$data_9 : \{data_8\}$

### III. RELIABILITY ANALYSIS AND TRACING

The functions  $Sources^L$  and  $\mathbb{I}^D$  allow us to trace back which parts of the system provide information to a certain service. This provides a basis for identifying of elementary services and the corresponding optimisation of the logical architecture. To trace which services are affected by the results produced by the service, we introduce another two functions,  $\mathbb{O}^D$  and  $Acc^L$ . For any service  $C$ , the function  $\mathbb{O}^D$  (dual to  $\mathbb{I}^D$ ) returns the corresponding set  $\mathbb{O}^D(C, x)$  of output channels depending on input  $x$ . This help us to solve following problems:

- if there are some changes in the specification, properties, constraints, etc. for  $x$ , we can trace which other channels can be affected by these changes;
- having for each output channel the minimal subservice computing the corresponding results, we can check the critical paths in the system by allocating to each subservice /output channel the corresponding worst case execution time (WCET). For this purpose we can use representation of the system architecture as a directed graph (cf. Section IV).

If the input part of the service's interface is specified correctly in the sense that the service does not have any "unused" input channels, the following relation will hold:

$$\forall x \in \mathbb{I}(C). \mathbb{O}^D(C, x) \neq \emptyset.$$

On each abstraction level  $L$  of logical architecture, we can define a function  $Acc^L : CSet^L \rightarrow (CSet^L \text{ set})$ . For any service (name)  $A$  the function  $Acc^L$  returns the corresponding (possibly empty) set of services (names)  $B_1, \dots, B_{AN}$  that are the acceptors for the output streams of  $A$ . This function dual to the function  $Sources^L$ :

$$x \in Acc^L(y) \Leftrightarrow y \in Sources^L(x)$$

This model allows us to analyse the influence of a service's failure on the functionality of the overall system: if a service  $C$

fails, the set of affected function will be  $Acc^L(C)$ . For example, for the system from Fig. 2, on the abstraction level  $L_1$  the services  $A_{12}$ ,  $A_{21}$ ,  $A_5$ , have no acceptors, where the set of acceptors of the service  $A_{11}$  is  $\{A_{21}, A_{22}, A_{23}, A_{31}, A_{32}, A_{41}\}$ . Thus, failure of the service  $A_{12}$  causes wrong behaviour only of  $A_{12}$  itself, where failure of the service  $A_{12}$  has influence on the results of seven services (including  $A_{12}$ ). On this basis, we can mark each service  $C$  by a number of services which are affected by  $C$  (and, respectively, by its failure). Thus, the *impact number* of the service  $A_{12}$  will be 1 (the minimal value), where the impact number of  $A_{11}$  will be 7. Moreover, on this basis we can apply results of our previous work on efficient hazard and impact analysis for automotive mechatronics systems [10], which was done in collaboration with industrial partners from ITK Engineering AG.

The representation of dependencies between services by a directed graph (like on Fig. 3) allows also to find the worst case execution time (WCET) needed for the concrete output based on the WCETs of the system's services, as well as analyse the influence of a service's failure on overall system. For example, from Fig. 3 it is easy to see that for the outputs of the services  $A_{11}$  and  $A_{12}$  the worst case computation time is equal to the WCETs of these services, for the output of the services  $A_{21}$  the worst case computation time is equal to the sum of WCETs of  $A_{11}$  and  $A_{21}$ , etc.

#### IV. STRONGLY CONNECTED SERVICES

After the decomposition discussed in the previous section, we obtain a (flat) architecture of system. The main feature of this architecture is that each output channel (within the system) belongs the minimal subservice of a system computing the corresponding results.

We represent this (flat) architecture as a directed graph and apply one of the existing distributed algorithms for the decomposition into its strongly connected services, e.g. FB [11], OBF [12], or the colouring algorithm [13]. For our goals, we extend them by a preliminary simplification of the graph (cf. below). This optimisation is algorithm independent and is also applicable for the case another decomposition is chosen.

Let us introduce some basic terms and definitions to explain the main ideas of the decomposition we apply in our approach. A *directed graph*  $G$  is a pair  $(V, E)$ , where  $V$  is a set of vertices, and  $E \subseteq V \times V$  is a set of directed edges. If  $(u, v) \in E$ , then  $v$  is called a *successor* of  $u$ , and  $u$  is called *predecessor* of  $v$ . A vertex  $t$  is called *reachable* from vertex  $x$  if  $(x, t) \in E^*$ , where  $E^*$  is a transitive and reflexive closure of  $E$ . If  $x_k$  is reachable from  $x_0$ , then there is a sequence of vertices  $x_0, \dots, x_k$  (called *path*), such that for all  $0 \leq i < k$   $(x_i, x_{i+1}) \in E$ .

A set of vertices  $SC \subseteq V$  is *strongly connected* if for any vertices  $u, v \in SC$  holds that  $v$  is reachable from  $u$ . A *strongly connected service (SCS)* is a maximal straggly connected set of vertices, i.e. after extension of this set by any additional vertex the set is no more strongly connected. An SCS is *trivial* if it consists of a single vertex. We call an SCS *leading (terminating) trivial* if it consists of a single vertex that has no predecessors (successors).

In our representation of a flat architecture as a directed graph, services become vertices and channels become edges. We distinguish here

- channels that are system's input/output (within the graph, we represent them by dashed edges, cf. also Fig. 3, because they are less important on this level of (de)composition – we do not need to take them into account by identifying the SCSs), and
- channels that are local for the system, i.e. representing dependencies between services.

Vertices that are leading trivial SCSs are labeled by LT, vertices that are terminating trivial SCSs are labeled by TT, and vertices that are *commonly trivial* SCSs (trivial, but neither leading nor terminating) are simply labeled by T. In addition, we define another special kind of trivial SCSs: services that has neither successors, no predecessors (this means that all their input/output channels are system's input/output channels). We call them *disconnected trivial* SCSs and label by DT.

Two vertices  $u$  and  $v$  of an undirected graph  $G$  are *connected* if there is a path from  $u$  to  $v$  (for the case of directed graph we need to ignore the orientation of its edges); otherwise, they are *disconnected*. A graph  $G$  is *connected* if every pair of vertices in  $G$  is connected; otherwise, it is called *disconnected*. If the graph  $GL_1$  representing a flat system's architecture is *disconnected*, i.e. consists of  $N$  graphs that are either connected graphs or single vertices (we denote this set by  $GN$ ), then we identify SCSs in each of these graphs separately and could also parallelise these computations (cf. also example below).

Thus, we start the service (de)composition by construction the set  $GN$  from the system's architecture on level  $L_1$ . In the case  $GL_1$  is *connected*, we simply have  $GN = \{GL_1\}$ . Then for each graph we proceed as follows:

- (1) We identify DT services and delete these vertices from the graph. This is non-recursive procedure with time complexity  $O(n)$ , where  $n$  is a number of vertices.
- (2) We apply the recursive elimination technique *One-Way-Catch-Them-Young* (OWCTY, [14]) to remove the LT services and the reversed OWCTY technique to remove the TT services.
- (3) The removed services become single services on the level  $L_2$ . To the rest of the graph, we apply one of the existing algorithms for the decomposition into SCCs (FB [11], OBF [12], etc.).

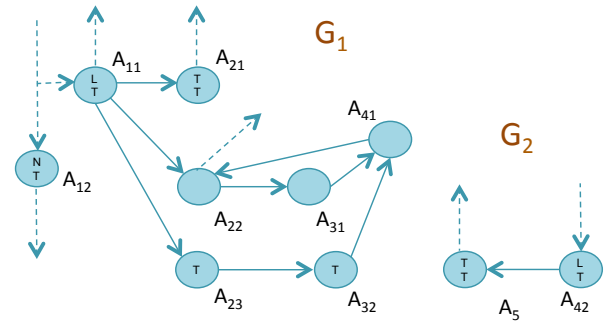


Fig. 3. System  $S$  (level  $L_1$ ): detection of the strongly connected services

To illustrate these ideas, we continue with the example from previous section. We represent the system from Fig. 2 as a directed graph to apply a decomposition algorithm, cf. Fig. 3.

This graph is disconnected and consists of two connected graphs,  $G_1$  and  $G_2$ , therefore we can analyse them in parallel. For  $G_1$  we need to perform the following steps:

(1) First of all, we identify  $A_{12}$  as a DT service, delete it from the graph (system's subservice  $S_1$  on the level  $L_2$ ).

(2) Then, we apply the OWCTY elimination techniques to identify and remove the LT and TT services:

- On the first run of the algorithm, we identify  $A_{11}$  and  $A_{21}$  as an LT and an TT services respectively and delete them. They become  $S_2$  and  $S_3$  on the level  $L_2$ .
- After this,  $A_{23}$  (which is commonly trivial in  $G_1$ ) becomes an LT service and should be deleted, becoming  $S_4$  on  $L_2$ . Then, the same situation will be with  $A_{32}$ : it should be deleted as an LT service and becomes  $S_5$ .
- The rest of the graph (vertices  $A_{22}$ ,  $A_{31}$ ,  $A_{41}$ ) with corresponding edges contains neither LT nor TT vertices. This is an input for the SCC-decomposition algorithm.

(3) In our example we apply a variant of the FB algorithm. The time complexity of this algorithm is  $O(n*(n+m))$ , where  $n$  is a number of vertices and  $m$  is the number of edges in this graph. The FB algorithm identifies that the vertices  $A_{22}$ ,  $A_{31}$ ,  $A_{41}$  build a single SCS, which becomes  $S_6$  on the level  $L_2$ .  $G_2$  has only two vertices, a leading trivial SCS  $A_{42}$  and a terminating trivial SCS  $A_5$ . Thus, on the level  $L_2$  it gives us two subservices of the system:  $S_7$  and  $S_8$ . Fig. 4 presents the result of the architecture optimisation.

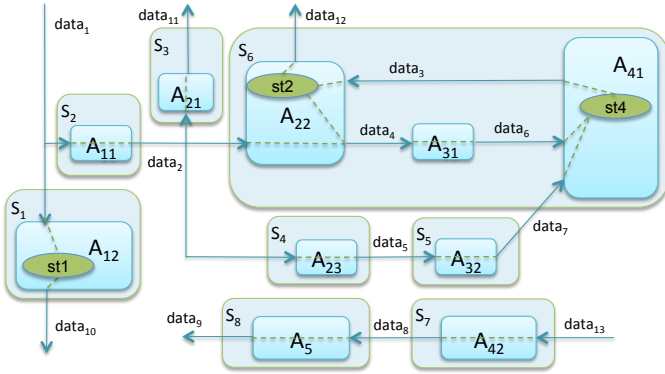


Fig. 4. Architecture of  $S$  (level  $L_2$ )

## V. EFFICIENT CHECKING OF PROPERTIES

A property can be represented by relations over data and control flows on the system's channels. Let for a relation  $r$ ,  $I_r$  and  $O_r$  be the sets of input and output channels of the system used in this relation. For each channel from  $O_r$  we recursively compute all the sets of the dependent input channels. Their union, restricted to the input channels of the system (for the case we exclude properties' specification over local channels),  $\mathbb{I}_r^D$  should be equal to  $I_r$ , otherwise we should check whether the property was specified correctly (i.e. exclude (human) error in the specification of the property, e.g., when a wrong channel identifier was used) and if so, precise it or eliminate unnecessary constraints:

(1) If a channel  $x$  belongs to  $\mathbb{I}_r^D$  but not to  $I_r$ , we need to extend  $r$  explicitly specifying that this property should hold for any values on  $x$ .

(2) If a channel  $x$  belongs to  $I_r$  but not to  $\mathbb{I}_r^D$ , this means that  $r$  is unnecessary strict, because it contains assumptions on irrelevant input channel.

From  $O_r$  we obtain the set  $OutComp_r$  of services having these channels as outputs, compute the union of corresponding sets  $Sources^{L_2}$ . This union together with  $OutComp_r$  give us the minimal part of the system needed to check the property  $r$ . This allowed us, especially in the case of cloud-supported processing, to reduce the costs of monitoring, testing or verification.

Assume we have to check relation  $r_1(data_{10}, data_{13})$  specified for the system presented on Fig. 4. It is easy to see, that there is no dependency between  $data_{13}$  and  $data_{10}$ . We have here  $I_{r_1} = \{data_{13}\}$  and  $O_{r_1} = \{data_{10}\}$ .  $data_{10}$  is a single output channel of  $S_1$ , i.e.  $\mathbb{O}(S_1) = \{data_{10}\}$ . This gives us  $OutComp_{r_1} = S_1$ . Thus,  $\mathbb{I}^D(S_1, data_{10}) = \{data_1\}$  which is already equal to  $\mathbb{I}_{r_1}^D$ , because  $data_1$  is a system input. First of all we need to exclude error in the specification of the property, e.g. the case where  $r_1$  is meant to be specified over  $data_{10}$  and  $data_1$ . If  $r_1$  was specified without such errors, we need (1) to extend it explicitly specifying that it should hold for any values on  $data_1$ , and (2) eliminate unnecessary constraints on  $data_{13}$  from  $r_1$ . Because  $Sources^{L_2}(S_1) = \emptyset$ , we need only the service  $S_1$  to check this property.

Assume now we have to check relation  $r_2(data_1, data_{12})$  specified for the same system. For this case  $I_{r_2} = \{data_1\}$  and  $O_{r_2} = \{data_{12}\}$ .

$data_{12}$  is a single output channel of  $S_6$ , and therefore  $OutComp_{r_2} = S_6$ .

$\mathbb{I}^D(S_6, data_{12}) = \{data_2, data_7\}$  (both channels are local),  $Sources^{L_2}(S_6) = \{S_2, S_5\}$ .

$data_2 \in \mathbb{O}(S_2)$ , which gives us  $\mathbb{I}^D(S_2, data_2) = \{data_1\}$  (this is a system input channel, therefore it should be a part of  $\mathbb{I}_{r_2}^D$ ).

$Sources^{L_2}(S_2) = \emptyset$ .

$data_7 \in \mathbb{O}(S_5)$ ,  $\mathbb{I}^D(S_5, data_7) = \{data_5\}$  (also local),  $Sources^{L_2}(S_5) = \{S_4\}$ .

$data_5 \in \mathbb{O}(S_4)$ ,  $\mathbb{I}^D(S_4, data_5) = \{data_2\}$  (local, cf. above),  $Sources^{L_2}(S_4) = \emptyset$ .

$\mathbb{I}_{r_2}^D = \{data_1\} = I_{r_2}$ , there is no need to extend the property or eliminate some of its constraints. To check this property we need the following set of services:

$OutComp_{r_2} \cup Sources^{L_2}(S_6) \cup Sources^{L_2}(S_2) \cup Sources^{L_2}(S_5) \cup Sources^{L_2}(S_4) = \{S_2, S_4, S_5, S_6\}$ .

## VI. REMOTE COMPUTATION

In the previous section we have analysed, which part of the overall information about the system and its input data is necessary to check the corresponding property. In the similar way, we can trace the influence of the system environment's properties on the properties of the system itself. However, the decision, which information need to be processed locally (local services) and which need to be sent to the cloud (remote services), should be made not only according to this analysis of a (logical) system architecture, but also taking into account the following aspects of data flows between services:



(1) measure for costs of the data transfer/ upload to the cloud  $UplSize(f)$ : size of messages (data packages) within a data flow  $f$  and frequency they are produced. This measure can be defined on the level of logical modelling, where we already know the general type of the data and can also analyse the corresponding service (or environment) model to estimate the frequency the data are produced;

(2) measure for requirement of using high-performance computing and cloud virtual machines,  $Perf(X)$ : complexity of the computation within a service  $X$ , which can be estimated on the level of logical modelling as well.

On this basis, we build a system architecture, optimised for remote computation. We associate an  $UplSize$  measure to each channel (to each data flow), and an  $Perf$  measure to each elementary service in the system (i.e. for any service on the abstraction level  $L_1$ ). A limit, above which a remote computation is desired, is denoted by  $HighPerf$ . A limit of the  $UplSize$  measure, above which a limited up-/download is desired, is denoted by  $HighLoad$ , i.e. if the source of the data is deployed locally then the receiving services should preferably also deployed locally, and i.e. if the source is deployed in the cloud then the receiving services should preferably also deployed in the cloud. We do not take into account costs of the transfer of software service themselves, assuming that this aspect is already covered by the  $Perf$  measure. The  $UplSize$  measure should be analysed only for the channels that aren't local for the services on abstraction level  $L_2$ .

Using graphical representation, we denote channels with  $UplSize \gg HighLoad$  by **thick red** arrows, and the services with  $Perf \gg HighPerf$  by **white** colour, where all other channel and services are marked **blue**. We use the same colouring notation when building the corresponding tables for the values above limits as well as when represented a system by the corresponding directed graph. For the compressed table representation, which allows readable representation of the large systems' measures, we omit the concrete values of the measures leaving only the notes on relations to the defined limits. While measuring  $Perf$  on the level  $L_1$ , we mark a service by a sum of  $Perf$  measures of its subservices on the level  $L_2$ . Thus, if the  $Perf$  measure of at least one subservice is high, the same hold for the service's  $Perf$ .

This analysis can also be done automatically, provided all the measures  $UplSize$  and  $Perf$  are defined for data and control flows as well as for (elementary) services. Using our approach, we can prove a number of properties for service composition according to these measures. We start with the architecture from abstraction level  $L_2$  and represent it as a directed graph. Then we simply remove edges that correspond to channels with low  $UplSize$ , and obtain a set of connected graphs. If any two vertices (or graphs) are "connected" by a single edge that is "split" and has no source vertex (this represents a situation where a system input goes to many systems's services), we see them as a connected graph too. Each of the graphs becomes a service on the abstraction level  $L_3$  represents a system architecture, optimised for remote computation.

Assume the following case for the system  $S$  from the previous examples (cf. also Fig. 4):

- channels with  $UplSize$  value  $\gg HighLoad$ :  $data_1, data_4, data_5, data_6, data_7, data_8$ .

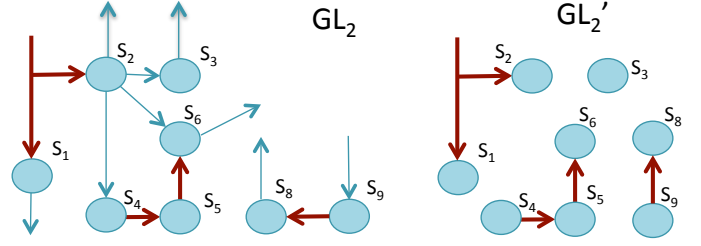


Fig. 5. Service (de)composition according to the channel measures  $UplSize$

- services with  $Perf$  value  $\gg HighPerf$ :  $A_{22}, A_{23}, A_{41}, A_{42}$ , and on the abstraction level  $L_2$ :  $S_2, S_4, S_6, S_7$ .

$data_4$  and  $data_6$  are local channels of  $S_6$ , their  $UplSize$  measure is not relevant for this case. We represent the system from the abstraction level  $L_2$  as a directed graph  $GL_2$  (cf. Fig. 5). After removing edges that correspond to channels with low  $UplSize$ , we obtain a set  $GL_2'$  of connected graphs.

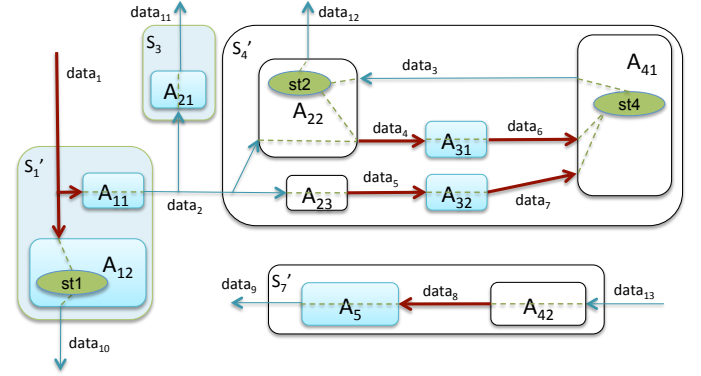


Fig. 6. Optimised architecture of  $S$  (Level  $L_3$ )

Fig. 6 represents a system architecture, optimised for remote computation. On the abstraction level  $L_3$  (cf. Fig. 6),  $S_1$  and  $S_2$  are composed together into a new service  $S_1'$ :  $data_1$  corresponds to a data flow, where messages (data packages) have large size and come with a high frequency, but  $data_2$  has very low frequency (e.g.,  $S_2$  realise a filtering function according to a given criteria), therefore it make more sense to deploy  $S_1$  locally, together with  $S_2$ .

There are no changes for service  $S_3$ , but  $S_4, S_5$ , and  $S_6$  are composed into  $S_4'$ , and  $S_7$  with  $S_8$  into  $S_7'$ . The services  $S_4'$  and  $S_7'$  have  $Perf$  measure higher  $HighPerf$ , therefore using high-performance computing and cloud virtual machines is required for these services.

## VII. SEMI-AUTOMATIC FORMAL VERIFICATION

To support this approach on verification level, we have also build a set of corresponding theories in Higher-Order Logic (HOL) using an interactive semi-automatic theorem prover (proof assistant) Isabelle [6]. This proof assistant is based on polymorphic HOL extended with axiomatic type classes, and support the proof of arbitrary mathematical theorems in interactive manner. Proofs are constructed in the structured proof language Isar [15]. The advantage of this proof language

is that the proofs are easy readable for both human and machine, which is not the case for many proof languages.

The representation of our approach in Isabelle/HOL contains approx. 70 general axioms and lemmas [16], which are necessary to analyse dependencies within a system using Isabelle in (semi-)automatic way, e.g. to verify whether a given set of services is equal to the set of (in)direct sources of some service, or whether the set of services is equal to the minimal set needed to check a certain property. To show a feasibility of the approach, we also have done a case study using our formalisation. In this case study [16], we started with 9 services connected with 24 channels on the abstraction level  $L_0$  (among them 4 services are specified using local variables). Fig. 7 presents how the number of services was changed among the abstraction levels.

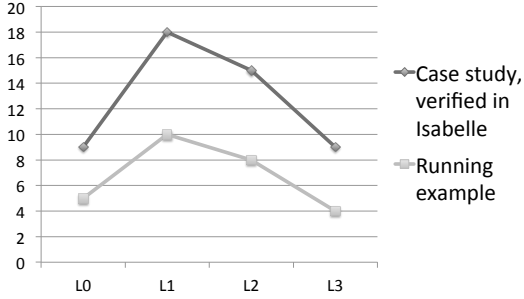


Fig. 7. Correlation between the number of services and abstraction levels: Case study and the running example (cf. Section II-B)

Overall, the case study contains more than 300 lemmas, approx. 50% of them can be composed and proven automatically, using the predefined schema, which is a part of our formalisation. All the technical details of formalisation within the theorem prover Isabelle as well as the corresponding proofs for a case study to elaborate the approach are presented in [16]. Approx. 90% of the proofs (general as well as for a concrete system from the case study) are constructed using Isabelle's component Sledgehammer [7]. Sledgehammer discharges proof goals applying to them resolution based first-order automatic theorem provers (ATPs) and satisfiability modulo theories (SMT), which makes the human-related part of the proof simpler and faster.

## VIII. DISCUSSION AND RELATED WORK

The approach presented above extends our previous work in [1], [2], [9] by providing (i) formal approach that allows modelling of (data and control flow) dependencies between components, and (ii) associated algorithm for architecture optimisation towards efficient verification, testing, and monitoring of system's properties. We leave here out of scope the automatization of the construction of the functions  $\mathbb{I}^D$ ,  $\mathbb{O}^D$ , etc. for a concrete case. For the example presented in this paper as well as for the case study, verified in Isabelle, these functions were specified manually. Thus, this automatization is planned for the future work. However, one of the advantages of our current work is formal semi-automatic analysis of (i) system architecture, and (ii) the sufficiency of the set of system's components to check a certain property. Thus, in our work we touch the following research areas: modelling

communication between components, system decomposition, as well as architecture modelling. In the rest of the section we discuss the related work on them.

*Modelling communication:* Various languages and techniques have been proposed to represent communication between components/processes, for example, Bergstra's Algebra of Communicating Processes [17], Hoare's approach on Communicating Sequential Processes [18] and its extension [19]. Magee et al. [20] tried to combine the ideas of operational semantics with Milner's  $\pi$  calculus, calculus of mobile processes (cf. [21], [22]). Reo, a channel-based coordination model for component composition, represents a co-algebraic view on this area [23], [24]. The work presented in [25] defines an extensive support to the components communication and time requirements, while the model discussed in [26] proposes general ideas on model for distributed automation systems.

Our approach, in contrast, is focusing not on the communication in general but on the dependency aspects and how their analysis can be used to increase the efficiency of properties checking.

*System decomposition.* There is a large number of approaches in the area of systems decomposition (see, e.g., [27], [28]). In general we can say, that decomposition in many cases leads to a refinement of a system, where by composing a system from components we can implicitly build a new level of abstraction of system representation (cf. [29], [30]).

The main difference and the main contribution of our current work, also compared with our previous work on formal decomposition [9], is (i) an extension of the specification approach to associate subspecifications with subservices or architectural components in a more usable, readable way, (ii) focusing on the aspects essential for the efficient checking of system's properties, (iii) taking into account aspects that are important for the case of using high-performance computing and cloud virtual machines. To this end, our focus in the current paper goes also beyond ideas presented in [9], where readability and manageability of specification were discussed in relation to inconsistencies and incomplete specifications.

*Architecture modelling.* An introductory overview of foundations and applications of the model driven architecture can be found in [31]. There is a large collection of approaches on architecture elaboration, with different aims and domain orientations, e.g., Medvidovic and Taylor [32] introduce a classification and comparison framework for software architecture description languages, Malek et al. [33] presents a framework for improving distributed system's architecture and their deployment, Broy et al. [8] focus on specification and design of services and layered architectures. A number of approaches, e.g., [34], propose several meta-models that introduce relationships between architectural design decision alternatives and activities related to them. A number of architecture description languages have been developed to specify compositional views of a system on an abstract level, e.g., TrustME [35], which combines software architecture specification approaches with ideas of design-by-contract and allows capturing of behavioural interaction patterns between large-scale components of software and systems architectures.

Our current work is focused on modelling of a logical service-oriented architecture. The main focus of our approach is on elaborating for a concrete system, which part of the

system is sufficient to check a certain property, and how to optimise the logical architecture towards efficient monitoring, testing, and verification of system's properties, also for the case of remote connection.

## IX. CONCLUSION AND FUTURE WORK

In this paper, we presented a formal approach to modelling and analysis data and control flow dependencies between components. We described the theory which lies behind the approach and presented a running example to illustrate the main ideas of the approach. We conclude that our approach allows

- to specify the dependencies within a system formally,
- to elaborate for a concrete system, which part of the system is sufficient to check a certain property,
- to analyse tracing and reliability aspects,
- to optimise the architecture of a given system, also taking into account such aspects as (i) costs of data transfer/upload to the cloud, and (ii) requirements of using high-performance computing and cloud virtual machines,

These results are especially important for analysis and optimisation of remotely deployed distributed control systems: our approach allows system decomposition oriented towards efficient checking of system properties – it allows to send to or from the cloud only the information really needed for the monitoring, testing or verification of the properties of interest.

Another contribution of our approach is a semi-automatic support of the presented ideas on verification level, an interactive semi-automatic theorem prover Isabelle/HOL.

*Future Work:* In the future work, we intent to automatise the construction of the functions that specify the dependencies ( $\mathbb{I}^D$ ,  $\mathbb{O}^D$ , etc.) for a concrete case on each level of abstraction. One of the other possible directions of our future work is on combination of the ideas presented above with our previous work on analysis of crypto-based components [36], to analyse the data dependencies between components wrt. secrecy/security properties.

## REFERENCES

- [1] M. Feilkas, F. Hölzl, C. Pfaller, S. Rittmann, K. Scheidemann, M. Spichkova, and D. Trachtenherz, "A Top-Down Methodology for the Development of Automotive Software," TU München, Tech. Rep. TUM-I0902, 2009.
- [2] M. Spichkova, "Architecture: Methodology of decomposition," TU München, Tech. Rep. TUM-I1018, 2010.
- [3] —, "Stream Processing Components: Isabelle/HOL Formalisation and Case Studies," *Archive of Formal Proofs*, 2013, Formal proof development.
- [4] —, *Design of formal languages and interfaces: "Formal" does not mean "unreadable"*, K. Blashki and P. Isaías, Eds. IGI Global, 2013.
- [5] —, "Human Factors of Formal Methods," in *In IADIS Interfaces and Human Computer Interaction 2012*. IHCI 2012, 2012.
- [6] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.
- [7] J. C. Blanchette, S. Böhme, and L. C. Paulson, "Extending Sledgehammer with SMT solvers," in *Automated Deduction*, ser. LNCS, N. Børner and V. Sofronie-Stokkermans, Eds., vol. 6803. Springer, 2011, pp. 116–130.
- [8] M. Broy, "Service-oriented Systems Engineering: Specification and design of services and layered architectures. The JANUS Approach," *Engineering Theories of Software Intensive Systems*, pp. 47–81, 2005.
- [9] M. Spichkova, "Architecture: Requirements + Decomposition + Refinement," *Softwaretechnik-Trends*, vol. 31:4, 2011.
- [10] S. Dobi, M. Gleirscher, M. Spichkova, and P. Struss, "Model-based hazard and impact analysis," TU München, Tech. Rep. TUM-I1333, 2013.
- [11] L. Fleischer, B. Hendrickson, and A. Põnar, "On identifying strongly connected components in parallel," in *Parallel and Distributed Processing*, ser. LNCS, J. Rolim, Ed. Springer, 2000, vol. 1800, pp. 505–511.
- [12] J. Barnat, J. Chaloupka, and J. van de Pol, "Improved distributed algorithms for scc decomposition," *Electron. Notes Theor. Comput. Sci.*, vol. 198, no. 1, pp. 63–77, 2008.
- [13] S. M. Orzan, "On distributed verification and verified distribution," Ph.D. dissertation, Free University of Amsterdam, 2004.
- [14] K. Fisler, R. Fraer, G. Kamhi, M. Vardi, and Z. Yang, "Is there a best symbolic cycle-detection algorithm?" in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, T. Margaria and W. Yi, Eds. Springer, 2001, vol. 2031, pp. 420–434.
- [15] M. Wenzel, "The Isabelle/Isar reference manual," 2013.
- [16] M. Spichkova, "Formalisation and analysis of component dependencies," *Archive of Formal Proofs*, 2014, Formal proof development.
- [17] J. A. Bergstra and J. W. Klop, "Algebra of communicating processes with abstraction," *Theor. Comput. Sci.*, vol. 37, pp. 77–121, 1985.
- [18] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.
- [19] G. Hilderink, "Graphical modelling language for specifying concurrency based on csp," *IEEE: Software*, vol. 150, pp. 108–120, 2003.
- [20] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proceedings of the 5th European Software Engineering Conference*. Springer, 1995, pp. 137–153.
- [21] R. Milner, *A Calculus of Communicating Systems*. Springer, 1982.
- [22] —, *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.
- [23] F. Arbab, "Reo: a channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, vol. 14, pp. 329–366, 6 2004.
- [24] S. Meng, F. Arbab, B. Aichernig, L. Astefanoaei, F. de Boer, and J. Rutten, "Connectors as designs: Modeling, refinement and test case generation," *Science of Computer Programming*, vol. 77, no. 7-8, 2012.
- [25] B. Vogel-Heuser, F. S., T. Werner, and C. Diedrich, "Modeling network architecture and time behavior of distributed control systems in industrial plant," in *37th Annual Conference of the IEEE Industrial Electronics Society*, ser. IECON, 2011.
- [26] T. Hadlich, C. Diedrich, K. Eckert, T. Frank, A. Fay, and B. Vogel-Heuser, "Common communication model for distributed automation systems," in *9th IEEE International Conference on Industrial Informatics*, ser. IEEE INDIN, 2011.
- [27] J. Philipps and B. Rumpe, "Refinement of Pipe-and-Filter Architectures," in *FM'99*, J. M. Wing, J. Woodcock, and J. Davies, Eds., no. LNCS 1708. Springer, 1999, pp. 96 – 115.
- [28] D. B. da Cruz and B. Penzenstadler, "Designing, Documenting, and Evaluating Software Architecture," TU München, Tech. Rep. TUM-I0818, 2008.
- [29] M. Broy, "Compositional refinement of interactive systems," *ACM*, vol. 44, no. 6, pp. 850–891, 1997.
- [30] M. Spichkova, "Refinement-based verification of interactive real-time systems," *Electronic Notes in Theoretical Computer Science*, vol. 214, pp. 131–157, 2008.
- [31] A. Rensink and J. Warner, Eds., *Model Driven Architecture - Foundations and Applications*, ser. LNCS, vol. 4066. Springer, 2006.
- [32] N. Medvidovic and R. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, 2000.
- [33] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An extensible framework for improving a distributed software system's deployment architecture," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 73–100, 2012.



- [34] J. Tyree and A. Akerman, "Architecture decisions: demystifying architecture," *Software, IEEE*, vol. 22, no. 2, pp. 19–27, 2005.
- [35] H. Schmidt, I. Poernomo, and R. Reussner, "Trust-by-contract: Modelling, analysing and predicting behaviour of software architectures," *J. Integr. Des. Process Sci.*, vol. 5, no. 3, pp. 25–51, Aug. 2001.
- [36] M. Spichkova, "Compositional properties of crypto-based components," *Archive of Formal Proofs*, 2014, Formal proof development.